1

Introducing IIS 7.0

COPYRIGHTED MATERIAL

Microsoft Internet Information Services (IIS) version 7.0 was introduced with the Windows Vista operating system as the main Windows web server. The same web server is going to be utilized by Windows Server 2008 with the same features, which means developing with Windows Vista IIS 7.0 will cost nothing when it is time to deploy on Windows Server 2008 IIS 7.0.

IIS 7.0 is a revolution in terms of web application processing and handling. It has been re-architected to provide a more robust, extensible, componentized web server that gives developers a better opportunity to integrate more into its features.

This chapter starts with an overview of new IIS 7.0 features. Application pools and worker processes are reviewed before diving into more advanced topics. The discussion goes deeper to cover the major components inside IIS 7.0. IIS 7.0 introduces the concept of modules as a new architectural design. Both native and managed modules are covered, with a brief description of each. The chapter ends by giving an overview on the request processing in IIS 7.0 and the new application pool modes: Integrated and Classic.

By the end of this chapter, you will have a good knowledge of the following:

- IIS 7.0 features overview
- Application pool and worker processes
- IIS 7.0 components
- Managed and native modules inside IIS 7.0
- IIS 7.0 request processing pipeline
- Integrated and Classic mode application pools

Overview of IIS 7.0

IIS 7.0 is the new web server that ships with Windows Vista and Windows Server 2008. Similar to the previous versions of IIS, this new version will continue to handle and process web requests that arrive at the Windows machine. The most mature version of IIS before the current one is IIS 6.0 which ships with Windows Server 2003. IIS 6.0 is very robust in terms of security, speed, process management, and reliability. IIS 7.0 builds its core engine on its predecessor and improves several areas. In addition, many new features have been added, making it extensible and manageable, thus leveraging IIS 7.0 to be a web server platform powerful enough to handle the challenges of present and future web applications.

The new IIS 7.0 features and characteristics are briefly summarized and presented in the next few sections to give a high-level overview of what has been done to improve the web server.

Modular Architecture

As mentioned above, IIS 7.0 bases its core engine on the best features of IIS 6.0 and adds to them the extensibility and accessibility for developers through its modular core engine. IIS 7.0 is based on a plug-in architecture that allows developers to have a hand in the processing of web requests. IIS 7.0 provides extensibility through its runtime pipeline, configuration management, and operational features to have a customizable web server for varying needs and requirements.

Making IIS 7.0 modular gives you the chance to customize it according to personal preferences and needs. Contrary to how the IIS 6.0 was configured, IIS 7.0 has most of its modules available but not installed. An administrator or developer can choose what modules or features to install and activate and what modules to deactivate. This provides both administrators and developers with a robust and reliable capability to configure the web server as needed. Figure 1-1 shows the new IIS 7.0 Manager listing the 40 available managed and native modules or features that ship with a full installation.

Figure 1-1

All modules are not installed by default, unless specified. Any module can be uninstalled and removed from the runtime pipeline processing, giving a flexible and dynamic experience in terms of choosing what to configure from built-in modules or even adding new modules and features. From the security point of view, an administrator or developer can choose what modules to include in the processing, hence affecting the overall performance of loading the configured modules to handle requests. This modular architecture helps reduce surface attacks by having the freedom to choose the modules to include and provides better performance by having the administrator or developer install only the required set of modules or features. IIS 7.0 managed and unmanaged modules are covered in detail later in this chapter.

Web server features or modules are configured through XML configuration files. The configuration files (discussed in a later section) are built into a hierarchy where at every level modules or features are configurable.

A Microsoft TechNet resource is available online that lists all the modules and features contained in IIS 7.0 and shows which modules are installed by default and which can be added later:
`http://technet2.microsoft.com/WindowsServer2008/en/library/0d35e92b-ddb7-4423-b1e5-df550e25713b1033.mspx`

Developing Modules and Features

The modular architecture introduced above discusses the ability to customize the modules installed on the web server whether by adding new ones or uninstalling existing ones. Adding new modules is easier with the new extensibility API for developing modules to integrate into IIS.

All of the native modules installed or shipped with IIS are developed on top of this extensibility API and this API is public, which means any developer can take that API and either redevelop an existing module or develop a new module as required.

The new extensibility API is built with C++ and it fully represents the new web server object model. The set of classes allows the developer to develop modules that can participate in request processing on IIS. This model is a replacement of the ISAPI extensibility model and is much easier to develop with since the new model includes a type-safe and well-encapsulated object model. Every needed web server object has a corresponding specialized object interface in the new API. For example, the `IHttpRequest` interface allows custom modules developed on top of the new extensibility API to access all the information related to the request under processing. The `IHttpResponse` interface allows custom modules to interact with the response generated for a request processed by IIS 7.0.

The new extensibility API even excels in terms of memory allocation and state management over ISAPI. In the days of ISAPI extensions, the developer had to take care of allocating and unallocating memory as required. The new extensibility API and most of the new IIS 7.0 APIs allocate server-managed memory for the data processed, which is different from the days of ISAPI extensions where developers had to take care of all the mess.

Finally, the new extensibility API allows modules to access features that were impossible to access before, such as request buffering and other IIS request processing tasks.

What about ASP.NET developers who are not ready to learn C++ to develop new modules for IIS? IIS 7.0 allows ASP.NET developers to utilize their existing ASP.NET module or create new ones using both the .NET 2.0 and 3.5 Frameworks and plug them automatically into the IIS request pipeline. In a later section, the ASP.NET integration process is explained in more depth.

Deployment and Configuration Management

IIS 7.0 uses a new configuration system that is conceptually much different from the IIS 6.0 centralized metabase configuration system. The new configuration system borrows many ideas from the current .NET 2.0 and 3.5 Frameworks configuration system, which is based on section groups and sections.

IIS 7.0 configuration system is based on XML configuration files mainly the `ApplicationHost.config` and `Administration.config` configuration files. Both of these files get deployed on the machine when IIS 7.0 is installed The configuration file of concern for most of the tasks related to IIS 7.0 is the `ApplicationHost.config` configuration file that contains all the new web server meta-data.

This configuration file contains global- and application-specific configuration sections. It resembles the .NET Frameworks configuration files: `machine.config` and the root `web.config` configuration files. The web server configuration file can be reached by browsing to the `%WINDIR%/System32/inetsrv/config` folder. Figure 1-2 shows the two main sections of the `ApplicationHost.config` configuration file.

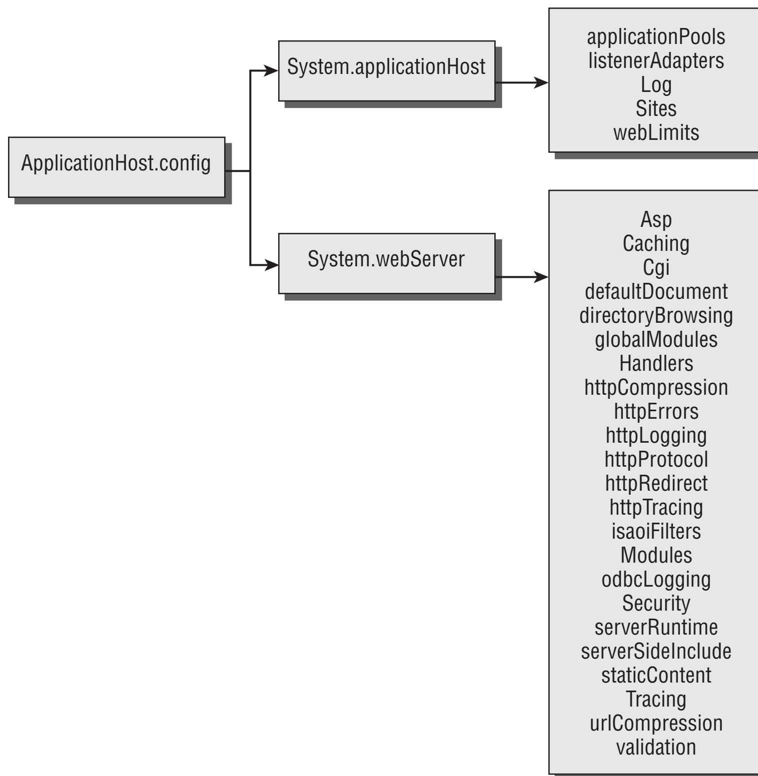

Figure 1-2

The two main section groups are the `<system.applicationHost>` and the `<system.webServer>` section groups. The `<system.applicationHost>` section group contains all the global settings for the web server, including the sites, applicationPools, listenerAdapaters, and so forth. This section is locked down and cannot be extended by any application hosted insideIIS.

```
        <sites>
            <site name="Default Web Site" id="1" serverAutoStart="true">
                <application path="/">
                    <virtualDirectory path="/" physicalPath="%SystemDrive%\inetpub\
wwwroot" />
                </application>
                <application path="/MyApp">
                    <virtualDirectory path="/" physicalPath="%SystemDrive%\inetpub\
wwwroot\MyApp" />
                </application>
                <bindings>
                    <binding protocol="http" bindingInformation="*:80:" />
                </bindings>
            </site>
```

The `<sites>` section defines all the configuration information on all sites hosted by the web server. At the root node there is the Default Web Site that points to the site located at `%SystemDrive%\inetpub\wwwroot`. To add a new website to IIS 7.0, simply add a new `application` node specifying the virtual `path` attributes together with a `virtualDirectory` sub-node setting the `path` and `physicalPath` attributes. With the above configuration, a new website has been added to IIS and can be accessed by `http://localhost/MyApp`.

The other section group, `<system.webServer>`, holds all the configurable sections for an application. For instance, this section contains configuration information about all the modules installed on the web server, a configuration section for directory browsing, and all the rest of the sections shown in Figure 1-2.

Note that with the new configuration system introduced by IIS 7.0, an administrator can configure the `<system.applicationHost>` and then select which section groups and sections from the `<system.webServer>` can be changed and edited by the application's `web.config` configuration file. This eliminates the need for a site owner to contact the administrator to change any settings in IIS, which was always happening before the release of IIS 7.0. This makes deployment with IIS 7.0 much easier. A developer can configure the `<system.webServer>` configuration section group during the development stage and then once the application is deployed, all the settings that were applied locally on IIS 7.0 would have the same effect on the hosting server given the fact that the administrator on the hosting server has already unlocked most of the configurable sections within the `<system.webServer>`. For instance, a developer can override the default web server settings for the default document for an application and set it to a customized page name.

```
<system.webServer>
    <defaultDocument>
        <files>
            <clear />
            <add value="MyPage.aspx" />
         </files>
    </defaultDocument>
</system.webServer>
```

The `<system.webServer>` configuration section group is the only section group in the `ApplicationHost.config` configuration file that can be extended and configured in the `web.config` configuration file of an application. The default documents configured on the web server are cleared out and a new customized default document for the current application is set to point to `MyPage.aspx`.

In regard to security, administrators are allowed to select which sections of the `<system.webServer>` to allow for editing and which are locked. For instance, an administrator can unlock many sections that do not pose any threat to the security of the web server as a whole and leave open all the sections that site owners usually require to change per application.

When a request reaches IIS for a resource, the different configuration files are joined together in a hierarchy to form single, unified configuration settings that apply to the current request. Figure 1-3 shows the process of how the different configuration files are grouped together to form a final `web.config` configuration file.

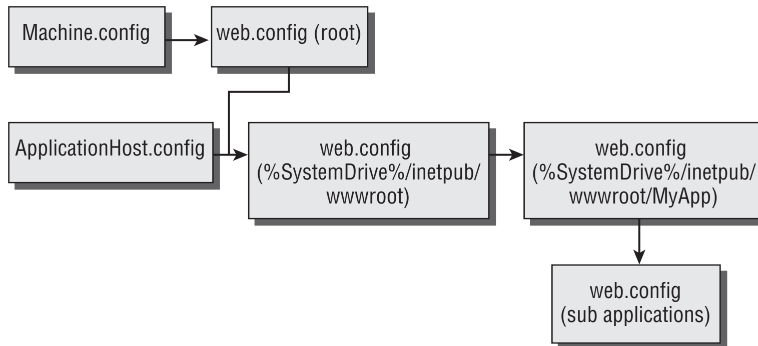

Figure 1-3

The `machine.config` file is merged with the `web.config` configuration file located in the root folder of the .NET 2.0 Framework, which is a shared folder used by both ASP.NET 2.0 and ASP.NET 3.5. The `ApplicationHost.config` configuration file is added to the result of the above grouping, and then the combined configuration settings are grouped with the `web.config` configuration file in the root website of the web server. The final result is added to the grouped configuration settings of the `web.config` configuration file of the executing application with its sub-applications' `web.config` configuration files.

An IIS resource is available online that gives a detailed overview of the `ApplicationHost.config` configuration file: `http://learn.iis.net/page.aspx/124/introduction-to-applicationhostconfig/`

Improved Administration

The IIS 7.0 Manager has been developed from scratch to replace the previous version. The difference is evident through the new UI experience and quick availability for any section to check and configure.

The IIS 7.0 Manager provides the UI interface experience for administrators and developers to configure the `ApplicationHost.config` configuration file without touching any physical resources. For instance, Figure 1-4 lists the available application pools in the `ApplicationHost.config` configuration file.

The Manager is just a UI representation to whatever is stored in the `ApplicationHost.config` configuration file. Using the manager to configure IIS 7.0 helps to prevent imposing possible wrong XML tag placement.

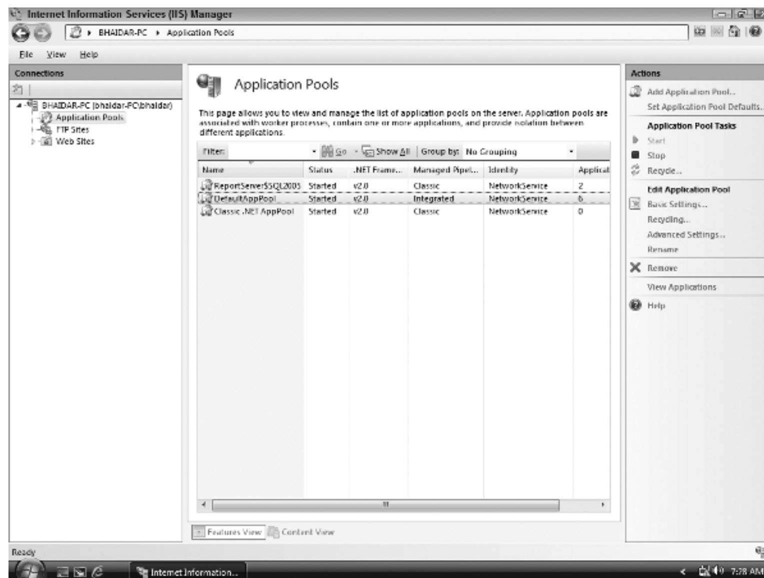

Figure 1-4

```
<applicationPools>
    <add name="DefaultAppPool" />
    <add name="Classic .NET AppPool" managedPipelineMode="Classic" />
    <applicationPoolDefaults>
        <processModel identityType="NetworkService" />
    </applicationPoolDefaults>
</applicationPools>
```

Application pools can be removed and edited, and new ones can be added. The result is stored in the `ApplicationPool` configuration section group inside the `ApplicationHost.config` configuration file.

The IIS 7.0 Manager inherits the idea of extensibility from IIS 7.0 and provides an extensible API that can be used to extend its UI features, hence extending the UI experience with much more features as required. In addition, the Manager allows management delegation that helps in administrating remote websites. For example, administrators in hosting companies can configure IIS 7.0 with the major and most secure configurations and allow the sites' owners to configure their sites remotely through their version of IIS 7.0 Manager. This does away with the need for special control panels for site owners to log into and configure their websites.

Moreover, the IIS 7.0 team thought of providing developers with a managed API to allow them to configure the IIS 7.0 configuration settings programmatically. The new API is called the `Microsoft.Web.Adminisration` API. Before this API can be used in Visual Studio, a reference has to be added to the `Microsoft.Web.Administration.dll` found at `%SystemDrive%:\Windows\System32\inetsrv`. The main class in this new API is the `ServerManager` .NET class. This class contains properties for the sites, applications, virtual directories, application pools, and worker processes.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Web.Administration;

namespace Microsoft.Web.Administration
{
    public class Program
    {
        static void Main(string[] args)
        {
            // Get a reference to the factory object
            // ServerManager
            var manager = new ServerManager();

            // Define a new website
            manager.Sites.Add(
                "ProgrammaticSite",
                @"D:\ProgrammaticSite\",
                8080);

            // Commit changes to the ApplicationHost.config
            manager.CommitChanges();
        }
    }
}
```

VB.NET

```
Imports System
Imports System.Collections.Generic
Imports System.Linq
Imports System.Text
Imports Microsoft.Web.Administration

Namespace Microsoft.Web.Administration
  Public Class Program
          Shared Sub Main(ByVal args() As String)
                 ' Get a reference to the factory object
                 ' ServerManager
                 Dim manager = New ServerManager()

                 ' Define a new website
                 manager.Sites.Add("ProgrammaticSite", "D:\ProgrammaticSite\",_
                 8080)

                 ' Commit changes to the ApplicationHost.config
                 manager.CommitChanges()
          End Sub
  End Class
End Namespace
```

The preceding code creates a new instance of the `ServerManager` factory object. Then it adds a new site by accessing the `Sites` property and specifying the site name, physical path, and the port, and finally, a call to the `CommitChanges` method to reflect the changes in the `ApplicationHost.config` configuration file. The result of executing the preceding code can be checked in the `<sites>` configuration section:

```
<site name="ProgrammaticSite" id="20">
    <application path="/">
        <virtualDirectory path="/" physicalPath="D:\ProgrammaticSite\" />
    </application>
    <bindings>
        <binding protocol="http" bindingInformation="*:8080:" />
    </bindings>
</site>
```

A new `site` entry is created within the `<sites>` configuration section group. The new site specifies the application's physical `path`, `virtualDirectory's` *`physicalPath`*, and the `protocol` binding.

Moreover, IIS 7.0 provides an additional tool called `appcmd.exe` that allows administrators and developers to configure the web server from the command prompt to create and configure sites, applications, virtual directories, start and stop application pools, recycle application pools, and much more. The utility is very rich in options and even presents a deeper configuration interface than that of IIS 7.0 Manager.

The book titled *Professional IIS 7 and ASP.NET Integrated Programming* (Wrox) explains in detail the IIS 7.0 Manager and the new Administration API. In addition, it includes informative chapters on the new IIS 7.0 configuration system and many more topics. An IIS resource is available online that gives a detailed overview of the Microsoft.Web.Administration API: `http://learn.iis.net/page.aspx/165/how-to-use-microsoftwebadministration/`

ASP.NET Integration

ASP.NET, since its release, has been used for several years to provide high level and powerful web applications developed purely within the context of the .NET Framework. A revolutionary stage has been introduced with the release of ASP.NET 2.0 that introduced new concepts and services to web development in ASP.NET. ASP.NET 3.5 continues to use the ASP.NET 2.0 at its core and adds to it additional new features and improvements to help developers build better and robust Web solutions.

So far, ASP.NET has been used only as a framework for developing dynamic web applications. IIS 7.0 leverages ASP.NET 2.0 and ASP.NET 3.5 to extensibility frameworks to extend the new web server.

IIS 6.0 handles requests for ASP.NET pages through ISAPI filters and extensions. Request handling is delegated to the ASP.NET ISAPI extension, the ASP.NET pipeline is triggered to handle the new request, and a response is generated and finally handed back to the IIS to deliver it to the requesting client. APS.NET has no control over what is being sent to its engine, since it is solely controlled by the IIS core engine. Only requests defined by the ASP.NET engine can be passed and processed, but what about other content? For instance, what if an ASP.NET application wants to secure access to some old Classic ASP pages using the same `FormsAuthenticationModule` used to protect ASP.NET resources? Before IIS 7.0, that was hard to do, if not impossible. If you are in a hurry to learn how to control and process non-ASP.NET content and resources through the ASP.NET pipeline, you can jump directly to

Chapter 7 for a detailed discussion on how to integrate ASP.NET security with Classic ASP pages. Note that whatever applies to Classic ASP applies also to any other non-ASP.NET resource including `.php`, `.jpg`, `.htm`, and so on.

In IIS 7.0, ASP.NET 2.0 and 3.5 can run in two different modes: Classic and Integrated. The Classic mode resembles the same model as that of IIS 6.0 and ASP.NET. ASP.NET 1.1 applications running inside IIS 7.0 can only be run using the Classic mode. When an ASP.NET 2.0 or 3.5 application is running in the Integrated mode, however, the ASP.NET engine gets unified with the IIS 7.0 engine, hence they share the same request pipeline. IIS's native C++ modules and ASP.NET `HttpModules` work together on processing a request. A request is processed by the configured native modules and any module registered with ASP.NET. One of the clear and shining results of this unified integration is that ASP.NET can now have a say when processing any content resource (and not only ASP.NET resources), a feature not present before the days of IIS 7.0. Figure 1-5 shows the unified request pipeline in processing a request in IIS 7.0.

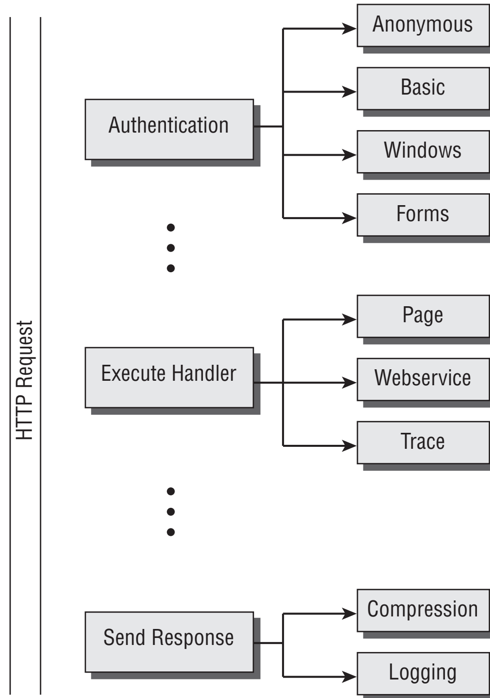

Figure 1-5

When it is time for IIS to authenticate a request, it executes all the configured native and managed authentication modules at the same time. The same applies for any stage inside IIS 7.0. This signifies again the power of having both ASP.NET modules and native modules execute side by side in handling a request.

More on ASP.NET integration with IIS 7.0 is covered in detail in Chapter 2.

Security Improvements

IIS 7.0 security is based on the robustness of IIS 6.0's security. By default, when IIS 6.0 is installed, it is installed in a locked-down mode, meaning that only handling of static files and the World Wide Web Publishing Service (WWW Service) are installed and enabled. The rest of services that operate on top of IIS 6.0 (including ASP, ASP.NET, and so forth) are disabled and can be added and enabled at any time by the administrator.

IIS 7.0 takes the locked-down strategy of IIS 6.0 one step further and follows the same locked-down pattern by installing fewer services at installation time. Having fewer features installed and enabled minimizes the risk of attack on the web server and minimizes the work done by the administrator to keep updating with patches and service packs on the different services installed, whether enabled or not. By making use of the modular architecture, an administrator can easily, at any time, install a new module or feature required by applications hosted by the web server.

Enabling the unified request pipeline in IIS 7 by configuring applications with the Integrated mode, the web server gains a more secure environment through the use of ASP.NET security modules. These modules include the `FormsAuthenticationModule` and the Membership and Role management services introduced early in ASP.NET 2.0 that still constitute a major feature in ASP.NET 3.5. Not only can ASP.NET benefit from these modules, but IIS 7.0 also gets better protection by utilizing these modules to protect the resources hosted in its environment.

In addition, IIS 7.0 introduces URL Authorization, which is inspired (more or less) by the architecture of the ASP.NET URL Authorization. The new authorization system allows administrators to add declarative access control rules for the hosted applications to protect their resources. This new feature integrates well with the ASP.NET Membership and Role management services. A more detailed discussion on URL Authorization is given in Chapter 3 of this book.

Moreover, the IIS 7.0 team replaced the old URL Scan security tool with a new `RequestFilteringModule` that gives administrators finer control on what to allow and disallow in a request targeting the web server. The `RequestFilteringModule`, as shown in the following code, can be configured through the `<system.webServer>` configuration section group either in the `ApplicationHost.config` configuration file or through the application's `web.config` configuration file.

```
<configuration>
 <system.webServer>
  <security>
   <requestFiltering>
    <fileExtensions allowUnlisted="false" >
     <add fileExtension=".aspx" allowed="true"/>
    </fileExtensions>
   </requestFiltering>
  </security>
 </system.webServer>
</configuration>
```

For instance, to configure IIS 7.0 to process ASP.NET web pages only, the `RequestFilteringModule` is configured to allow only ASP.NET web pages and prevent all other file extensions from being served and processed.

For further details, an IIS resource is available online that gives a wider overview of the new `RequestFilteringModule`: `http://learn.iis.net/page.aspx/143/how-to-use-request-filtering/`

Another security feature in which IIS 7.0 excels is the IIS Manager. As mentioned above, when applications are hosted locally, the site owner can configure IIS 7.0 settings by either direct access to the `ApplicationHost.config` configuration file, or through the `appcmd.exe` command-line utility, or programmatically by utilizing the `Microsoft.Web.Adminsitration` API. When configuring remote applications, IIS Manager provides remote connections to site owners through their local instance of the manager through firewall-friendly HTTPs connections. Based on the restrictions set by the remote administrator, a site owner connects to the remote web server through the local instance of the manager. The user gets authenticated on the remote server either by Windows authentication, if the user has a Windows account on the remote server, or by custom authentication of the ASP.NET Membership services. Once authenticated, the site owner can now configure the web server's settings under the limitations set by the remote administrator.

Not only does IIS Manager allow remote connections; it also allows administrators to configure the IIS Manager UI to select the features to show for remote connections. This is yet another security protection on the hosting web server.

Finally, IIS 7.0 introduces a new IIS anonymous account, the `IIS_USR`. This built-in account has no expiration date, nor does it need any password synchronization among different machines. Also, a new group is `IIS_IUSRS` that replaces the old `IIS_WPG` group. This group injects itself into the identity of the Worker process automatically at runtime. This makes the process of specifying another custom account for the Worker process identity easier without having to worry about adding this custom account to the `IIS_IUSRS` group. Since the `IIS_IUSR` and `IIS_IUSRS` are built-in, any Windows access control lists (ACLs) that an administrator or developer assigns on one machine can be copied to another machine, for instance, from the development machine to the testing and deployment machine, without any further worries, making the deployment process easier and more flexible.

Troubleshooting Improvements

IIS modular architecture not only introduces flexibility and robustness in configuring the web server, but it also adds more complexities when it comes to debugging or tracing requests when a problem occurs while a resource is being executed by the web server. Therefore, several new troubleshooting improvements have been added to allow administrators and developers to better detect what is going wrong with their applications.

A new, improved tracing system is added to the IIS infrastructure that is capable of capturing all related information for a request being processed by the web server. This way, an administrator can refer back at any time to check the status of requests being served by IIS. The trace information generated by the web server can be monitored and listened to by a new feature, the Failed Request Tracing feature. This new feature is basically configured to listen only to *failure* requests and logs them to the hard-disk. Before using this feature, it must be enabled in the IIS Manager tool. Figure 1-6 shows how to open up the Failure Request Tracing form to enable/disable the feature and to specify the path where to log the trace data.

By default, the Failure Request Tracing feature passes all successful requests and logs only the failed ones, as mentioned above. In addition, an administrator can define Failure Request Tracing Rules to specify what trace information to listen to in the web server tracing system. To define these rules, the Failing Request Tracing Rules feature can be configured inside the IIS Manager tool reached by selecting Server Name ➪ Web Sites ➪ Default Web Site ➪ Failed Request Tracing Rules under the IIS section.

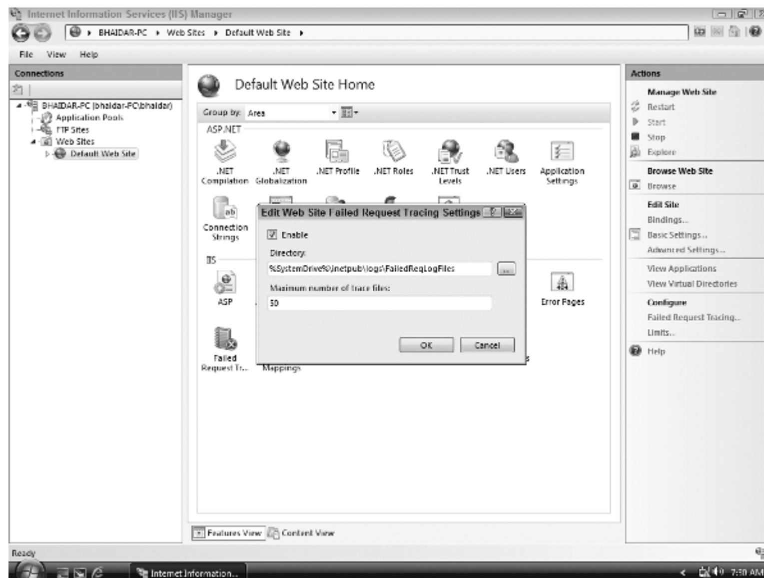

Figure 1-6

In addition, IIS provides new error information pages when errors are detected in the resources being processed. These error pages are similar in concept to the error pages generated by ASP.NET when an exception or error occurs in the application while a request is being made to any of its resources. The IIS error information pages give details about the problem that occurred, what module caused the problem, if any, where to find more tracing information about the specific failure of the request, and even more information that helps the administrator or developer to locate the problem quickly. The detailed error pages are configured for local access only by default and can be localized for any culture of preference.

To better benefit from the unified integration model between IIS and ASP.NET, the new web server's tracing system exposes its functionality to the modules created by the managed code in ASP.NET. The new tracing system is extensible enough to allow the managed modules registered in IIS to make use of the tracing information and to emit tracing data to the IIS tracing system. ASP.NET 2.0 and ASP.NET 3.5 contain the `System.Diagnostics.TraceSource` class that makes the developer's life easier in handling tracing events, data, and information (shown in the following code). The tracing system present in IIS 7.0 integrates with the tracing system in ASP.NET 2.0 and 3.5, thus allowing tracing information generated by ASP.NET to flow to the IIS 7.0 tracing system.

C#

```
using System;
using System.Diagnostics;
using System.Web;

public class CustomTracing : IHttpModule
{
    // Private member to hold a reference to the
    // TraceSource class
    private TraceSource tsTracing;
```

```
        /// <summary>
        /// Initialize event in the HttpModule
        /// </summary>
        /// <param name="application"></param>
        public void Init(HttpApplication application)
        {
            // Attach to the EndRequest event
            application.EndRequest += new EventHandler(application_EndRequest);

            // Define the trace source
            tsTracing = new TraceSource("tsTracing");
        }

        /// <summary>
        /// Handles the end request event
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        void application_EndRequest(object sender, EventArgs e)
        {
            // Write a message to the configured trace listeners mentioning the start of
            // a logical operation  or event, which is in this case beginning of the
            // EndRequest method.
            this.tsTracing.TraceEvent(
                TraceEventType.Start,
                0,
                "[CustomTracing MODULE] START EndRequest");

            // Get a reference to the HttpContext
            var app = (HttpApplication)sender;
            var context = app.Context;

            // Write some text to the response stream
            context.Response.Write(
                "Testing Tracing from ASP.NET and integrating into IIS 7.0");

            this.tsTracing.TraceEvent(
                TraceEventType.Verbose,
                0,
                "A debugging trace message to the trace listener!");
            this.tsTracing.TraceEvent(
                TraceEventType.Critical,
                0,
                "A fatal error or crash message to the trace listener!");
            this.tsTracing.TraceEvent(
                TraceEventType.Error,
                0,
                "A recoverable error message to the trace listener!");
            this.tsTracing.TraceEvent(
                TraceEventType.Information,
                0,
                "An informational message to the trace listener!");

            // Write a message to the configured trace listeners mentioning the end of a
            // logical operation or event, which is in this case end of the EndRequest
            // method
```

```
            this.tsTracing.TraceEvent(
                TraceEventType.Stop,
                0,
                "[CustomTracing MODULE] STOP EndRequest");
        }


        #region IHttpModule Members

        public void Dispose()
        {
            throw new NotImplementedException();
        }
        #endregion
    }
```

VB.NET

```
Imports System
Imports System.Diagnostics
Imports System.Web

Namespace CustomTracingModule
    Public Class CustomTracing
        Implements IHttpModule
        ' Private member to hold a reference to the
        ' TraceSource class
        Private tsTracing As TraceSource

        ''' <summary>
        ''' Initialize event in the HttpModule
        ''' </summary>
        ''' <param name="application"></param>
        Public Sub Init(ByVal application As HttpApplication) Implements_
        IHttpModule.Init
            ' Attach to the EndRequest event
            AddHandler application.EndRequest, AddressOf application_EndRequest

            ' Define the trace source
            tsTracing = New TraceSource("tsTracing")
        End Sub

        ''' <summary>
        ''' Handles the end request event
        ''' </summary>
        ''' <param name="sender"></param>
        ''' <param name="e"></param>
        Private Sub application_EndRequest(ByVal sender As Object,_
        ByVal e As EventArgs)
            ' Write a message to the configured trace listeners
            ' mentioning the start of a logical operation
            ' or event, which is in this case beginning of the EndRequest method.
            Me.tsTracing.TraceEvent(TraceEventType.Start,_
                               0,_
                               "[CustomTracing MODULE] START EndRequest")
```

```
                ' Get a reference to the HttpContext
                Dim app = CType(sender, HttpApplication)
                Dim context = app.Context

                ' Write some text to the response stream
    context.Response.Write("Testing Tracing from ASP.NET and integrating into IIS 7.0")

                Me.tsTracing.TraceEvent(TraceEventType.Verbose,_
                               0,_
                               "A debugging trace message to the trace listener!")
                Me.tsTracing.TraceEvent(TraceEventType.Critical,_
                               0,_
                               "A fatal error or crash message to the trace listener!")
                Me.tsTracing.TraceEvent(TraceEventType.Error,_
                               0,_
                               "A recoverable error message to the trace listener!")
                Me.tsTracing.TraceEvent(TraceEventType.Information,_
                               0,_
                               "An informational message to the trace listener!")

                ' Write a message to the configured trace listeners
                ' mentioning the end of a logical operation
                ' or event, which is in this case end of the EndRequest method
                Me.tsTracing.TraceEvent(TraceEventType.Stop,_
                               0,
                               "[CustomTracing MODULE] STOP EndRequest")
            End Sub


    #Region "IHttpModule Members"

            Public Sub Dispose() Implements IHttpModule.Dispose
                Throw New NotImplementedException()
            End Sub
    #End Region
        End Class
    End Namespace
```

The preceding code defines a local instance of the `TraceSource` class to hold all the tracing information by the managed ASP.NET module. The name of the `TraceSource` is important, as it will be referenced later as a source for the IIS trace listener. The `HttpModule` subscribes to the `EndRequest` event of the module and writes some dummy text into the response stream. Several trace messages have been written to the ASP.NET tracing system using the `TraceSource` object. Several methods are available in the aforementioned object, one of which is the `TraceEvent` method that takes as one of the inputs a value from the `TraceEventType` enumeration that defines the purpose of the trace message and another input, the trace message to be sent to the trace listener. There are several values in the `TraceEventType` enumeration that defines the different contexts in which a trace message might be present.

.NET 3.5 Framework ships with the `System.Web.IisTraceListner` class, which is used to route tracing information from ASP.NET tracing system to the IIS tracing infrastructure. To define the trace listener and attach it as a listener to the `TraceSource`, the `<system.diagnostics>` configuration section in the `web.config` configuration file is used.

```
<system.diagnostics>
    <sharedListeners>
      <add name="IisTraceListener" type="System.Web.IisTraceListener, System.Web,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
    </sharedListeners>
     <switches>
      <add name="DefaultSwitch" value="All" />
    </switches>
    <sources>
      <source name="tsTracing" switchName="DefaultSwitch">
        <listeners>
          <add name="IisTraceListener" type="System.Web.IisTraceListener, System.
Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
        </listeners>
      </source>
    </sources>
  </system.diagnostics>
```

The preceding configuration section defines the new IIS trace listener with a switch to capture all tracing information. In addition, the tracing source, which is in this case the `TraceSource` instance defined previously in the custom tracing managed module, is added and configured with the `IISTraceListener`. The preceding configuration section makes sure all the tracing information from ASP.NET is routed correctly to the IIS tracing system. The Failed Request Tracing feature can then be used, either through the default behavior to capture only failure trace information for failing requests or by adding custom rules to capture specific tracing information descending from the ASP.NET tracing system.

Finally, native developers can now troubleshoot the state of the IIS web server through the new Runtime Status and Control (RSCA) API known as "reeska." This new API allows native developers, mainly C++ developers, to examine the real-time status of the server by checking the active states of the sites and application pools, the running worker processes, and even to check current requests that are being processed. Developers can check the normal flow of page execution on the server and identify bottlenecks, while the different modules take their part in the request processing in the IIS pipeline. In addition, RSCA provides a means to control the state of the web server by stopping and starting the service, recycling application pools, starting and stopping sites, etc. These features are similar to the `appcmd.exe` command-line tool mentioned previously in this chapter.

An IIS resource is available online that gives an overview on developing managed tracing modules and routing the ASP.NET trace information to the IIS 7.0 tracing system: `http://learn.iis.net/page.aspx/171/how-to-add-tracing-to-iis-7-managed-modules/`

Application Pools

IIS 6.0 introduced the concept of application pools when operating in the worker process isolation-mode compared to working in the IIS 5 mode. An application pool by definition is a unit of separation, at the web server level, that is used to logically group applications into different boundaries, hence providing an isolation of execution from one application to another. If an application in one of the application pools on the web server crashes, not all the applications on the web server will be crashed too. This is because if each application is assigned to a separate application pool, then only this specific application

pool will recycle and all applications assigned to the same application pool will also crash. Other applications assigned to other application pools continue to function properly as if nothing happened on the web server. Therefore, application pools provide isolation of execution under the boundaries of the server resources allocated to every application pool, which are allocated differently from one application pool to another.

In the previous release of IIS, the web server was configured to either run in the worker process isolation mode or in the IIS 5.0 mode. However, in IIS 7.0, an application pool is created and its managed pipeline mode property is either set to *Integrated mode* or *Classic mode*. This means that the managed pipeline mode is not configured on the web server as a whole. On the contrary, several application pools can be created on IIS 7.0 with different managed pipeline modes, and applications can be assigned to any of those application pools, hence it is possible to run applications on the same web server with different modes of execution. Figure 1-7 shows the basic settings window for any application created inside IIS 7.0.

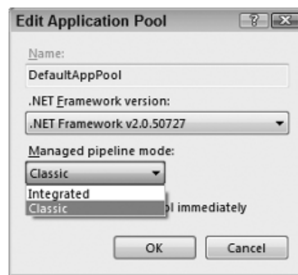

Figure 1-7

By opening the IIS manager tool, on the Actions tab on the right of the manager, there is a link to view application pools. All of the application pools created on the web server are listed. Right-clicking any of the application pools and selecting basic settings yields the screen shown in Figure 1-7. There is nothing special about it, but the managed pipeline mode combo box that allows you to choose either the Integrated or Classic mode.

Integrated Mode

When an ASP.NET 2.0 or 3.5 application is assigned to an application pool running in the Integrated mode, the application will benefit from the IIS and ASP.NET unified request processing pipeline. This means the request is processed by both the native and managed installed modules and ASP.NET will have the ability to process all types of content within that specific application. This mode is recommended when there is a need to execute an application in the Integrated mode, and it is the preferred mode to configure the application pools. Several additional and advanced settings can be set by right-clicking on the specific application pool and selecting Advanced Settings.

Classic Mode

The Classic mode resembles an IIS 6.0 application pool when the web server is running in a worker process isolation-mode. In IIS 7.0, applications are still given the opportunity to function as if they are being served by IIS 6.0. When an application is assigned to an application pool configured to run in the Classic mode, IIS 7.0 handles the execution of the application in the same way as IIS 6.0. For instance, if

an ASP.NET application is assigned to function under an application pool configured with Classic mode, the default and only available option for ASP.NET 1.1 application, when a request reaches IIS for that application, only the native modules will be executed on the request, then IIS 7.0 hands the request to the `aspnet_isapi.dll` extension to be processed by the ASP.NET runtime. Hence, IIS is able to process the request with all the installed native modules and ASP.NET will have another round in executing its managed modules; the same old-fashioned way of executing applications under IIS 6.0 when configured to run in the worker process isolation mode. If any ASP.NET application for some reason cannot run inside the application pool Integrated mode, it is recommended to keep it configured with the Classic mode under IIS 7.0. It will be executed and processed as if it is hosted in an IIS 6.0 environment.

IIS 7.0 Components

IIS 7.0 is made up of several components that form the web server internal core engine. These components include protocol listeners, services such as the w3svc service and the WAS service, protocol adapters, and many more core components. This section will present an overview of some of the protocols and services that handle request processing inside IIS 7.0.

Protocol Listeners

Protocol listeners are services in which each service is configured to listen and process a specific protocol request coming from the network on which the machine hosting the web server resides. For instance, one of the listeners installed on a Windows machine keeps on waiting and listening for any web request arriving on the machine. There are additional listeners also present to listen to other, different protocols. When a request is received by a listener, it forwards it to IIS 7.0 to be processed. Once a request is processed by IIS 7.0, the response generated is sent back to the protocol listener that originally sent the request. Finally, the response is handed back to the requestor.

An example of a protocol listener is the HTTP listener called Hyper Text Protocol Stack. This is the main protocol listener for all HTTP requests arriving on a Windows machine. When an HTTP request is first received by Windows Vista or Windows Server 2008, the initial handling is actually performed by the kernel-mode HTTP driver: `http.sys`.

World Wide Web Publishing Service

In IIS 6.0 the WWW service was responsible for several tasks at once. These tasks included HTTP administration and configuration, process management, and performance monitoring. In IIS 7.0, this has changed and the WWW Service now acts as a listener adapter for `http.sys`. A listener adapter is responsible for configuring the `http.sys` protocol listener with the IIS 7.0 configuration information stored in the `ApplicationHost.config` configuration file. It then waits for changes in the configuration information to reflect them into the `http.sys`, and finally notifies the Windows Process Activation Service (WAS) when a new HTTP request enters the local queue.

WWW Service functionality has been split into other services. It has preserved its role as a listener adapter, however, the rest of its responsibilities have been passed into another service called the Windows Process Activation Service.

Windows Process Activation Service

In IIS 7.0, the WAS is the second half of the WWW service that was present in the IIS 6.0 days. The WAS is a new service that has three main parts. Figure 1-8 shows the architecture and main components of the WAS.

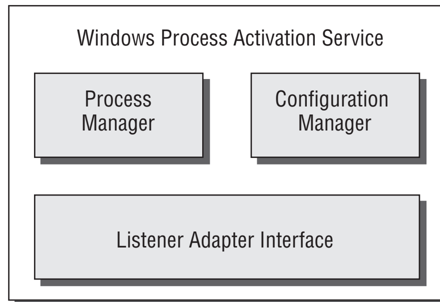

Figure 1-8

The configuration manager is responsible for reading the configuration information from the `ApplicationHost.config` configuration file. This manager reads global configuration information and protocol configuration information for both HTTP and non-HTTP protocols in order to be able to configure all protocol listeners installed on the web sever machine. It also reads application pool configuration information to know what application pools are present when processing requests on the server. It reads site configuration information, including the different applications included in each site together with the bindings defined on each application, and finally, reads the application pool each application belongs to. Such information helps the WAS when processing a request to know which site and application the request belongs to so that it gets handled by the right application pool.

In addition, the configuration manager gets a notification when the `ApplicationHost.config` configuration file changes so that it updates its data with the new ones and reflects this on the available protocol listeners.

The process manager is responsible for managing the application pools and worker processes for both HTTP and non-HTTP requests. It manages the state of the application pool by stopping, starting, and recycling it. In addition, when WAS receives a new request from one of the configured protocol listeners, it determines to which application the request belongs. It then checks with the configuration manager for the application pool of the application that the current request belongs to. Once the application pool is determined, it checks to see if there is any worker process currently active. If it finds one, it sends the request to the application pool to be processed by the worker process. If there is no worker process active inside the application pool, WAS instantiates a new one to process the current and upcoming requests.

The last component of the WAS is the unmanaged listener adapter interface. This layer inside the WAS defines how the external listeners communicate the requests they receive into the WAS in order to process them by the web server.

On startup of IIS 7.0, WAS gets initiated and performs several tasks. Figure 1-9 shows the flow of interaction when WAS first configures the protocol listener adapters.

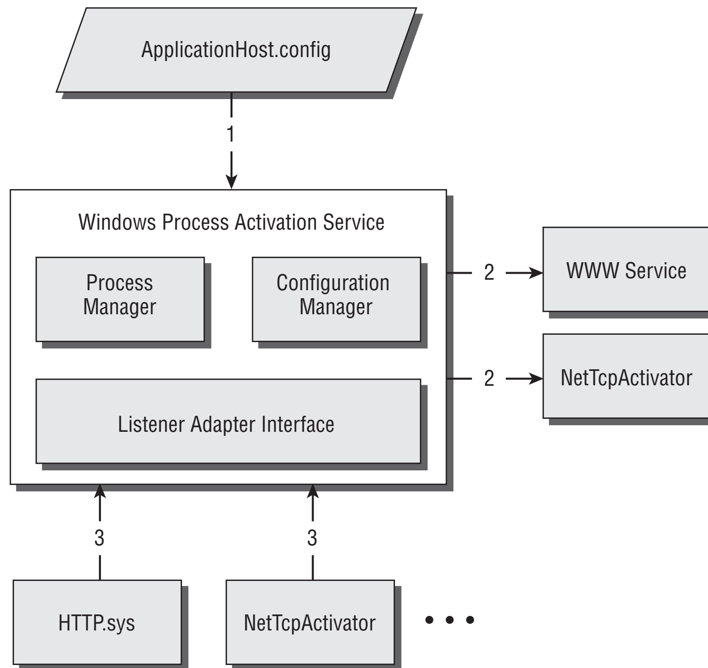

Figure 1-9

When WAS is instantiated, it first reads the configuration data from the `ApplicationHost.config` configuration file. Once the configuration information is read, it interacts with the configured protocol listener adapters to pass to them the needed configuration information. Protocol listener adapters function as the glue between the WAS and the protocol listeners. For instance, the WAS passes the configuration information into the WWW Service, the `http.sys` protocol listener adapter, which in turn configures `http.sys` to start listening for HTTP requests.

Once a new request comes in, the specific protocol listener communicates the request to the WAS through the listener adapter interface, so that the request gets processed. Once a response is ready for the request, WAS passes the response back to the protocol listener responsible for delivering the response back to the client. Again, WAS uses the listener adapter interface for the incoming and outgoing communication with the protocol listeners.

As shown in Figure 1-9, `NetTcpActivator` is the protocol listener and adapter for handling WCF requests. This indicates that WAS can process HTTP and non-HTTP requests; that means WAS can function properly without the need for the WWW Service by serving only non-HTTP requests. A good MSDN resource on the WCF listener adapters and hosting WCF applications inside IIS 7.0 is available online at `http://msdn2.microsoft.com/en-us/library/ms730158.aspx`

IIS 7.0 Modules

The modular architecture of IIS 7.0 has been discussed thoroughly at the beginning of this chapter. It is the new architecture that characterizes the web server core engine. Modules or features can be thought of as classes or objects embedding certain functionality that get executed whenever a new request is being processed by the IIS pipeline. Every installed module gets its turn in processing every request entering the IIS 7.0 pipeline.

This modular architecture has several goals, but above all it protects the web server from security attacks. When a small number of modules are installed on the web server, this means there is a lower probability for a security attack on the server, hence lowering the surface attack to hackers. In addition, when a small number of modules are installed, this means less security patches and updates are required for the administrator to maintain. Moreover, being able to customize the web server to this extent gives the administrator the chance of deciding on the role of the web server by installing and uninstalling modules in the way best suited for the role intended for the web server.

IIS 7.0 ships with a set of unmanaged or native modules that are all installed in case of a full installation of the web server. In addition, IIS 7.0 allows you to extend its functionality with managed modules. Each of these modules is discussed in detail.

Unmanaged Modules

The native modules are grouped by functionality. There are HTTP-related modules that perform tasks specific to HTTP; another set of modules perform tasks related to security; and anther set of modules perform tasks related to content (static files, directory browsing, and so on). There are a set of modules responsible for compression, modules concerned with caching, modules responsible for logging and diagnostics, and modules that help in integrating managed modules. All of these modules are fired and executed during the request-processing pipeline. The available native modules at the time of this writing together with a description are listed in the following table.

Module Name	Description
HTTP Modules	
`CustomErrorModule`	Sends default and configured HTTP error messages when an error status code is set on a response.
`HttpRedirectionModule`	Supports configurable redirection for HTTP requests.
`OptionsVerbModule`	Provides information about allowed verbs in response to OPTIONS verb requests.
`ProtocolSupportModule`	Performs protocol-related actions, such as setting response headers and redirecting headers based on configuration.
`RequestForwarderModule`	Forwards requests to external HTTP servers and captures responses.

Module Name	Description
TraceVerbModule	Returns request headers in response to TRACE verb requests
Security Modules	
AnonymousAuthModule	Performs Anonymous authentication when no other authentication method succeeds.
BasicAuthModule	Performs Basic authentication.
CertificateMappingAuthenticationModule	Performs Certificate Mapping authentication using Active Directory.
DigestAuthModule	Performs Digest authentication.
IISCertificateMappingAuthenticationModule	Performs Certificate Mapping authentication using IIS certificate configuration.
RequestFilteringModule	Performs URLScan tasks, such as configuring allowed verbs and file extensions, setting limits, and scanning for bad character sequences.
UrlAuthorizationModule	Performs URL authorization.
WindowsAuthModule	Performs NTLM integrated authentication.
Content Modules	
CgiModule	Executes CGI processes to build response output.
DavFSModule	Sets the handler for Distributed Authoring and Versioning (DAV) requests to the DAV handler.
DefaultDocumentModule	Attempts to return the default document for requests made to the parent directory.
DirectoryListingModule	Lists the contents of a directory.
IsapiModule	Hosts ISAPI extension DLLs.
IsapiFilterModule	Supports ISAPI filter DLLs.
ServerSideIncludeModule	Processes server-side includes code.
StaticFileModule	Serves static files.
FastCgiModule	Supports FastCGI, which provides a high-performance alternative to CGI.

Continued

Module Name	Description
Compression Modules	
`DynamicCompressionModule`	Compresses responses, and applies Gzip compression transfer coding to responses.
`StaticCompressionModule`	Performs precompression of static content.
Caching Modules	
`FileCacheModule`	Provides user-mode caching for files and file handles.
`HTTPCacheModule`	Provides kernel-mode and user-mode caching in http.sys.
`SiteCacheModule`	Provides user-mode caching of site information.
`TokenCacheModule`	Provides user-mode caching of user name and token pairs for modules that produce Windows user principals.
`UriCacheModule`	Provides user mode caching of URL information.
Logging and Diagnostics Modules	
`CustomLoggingModule`	Loads custom logging modules.
`FailedRequestsTracingModule`	Supports the Failed Request Tracing feature.
`HttpLoggingModule`	Passes information and processing status to http.sys for logging.
`RequestMonitorModule`	Tracks requests currently executing in worker processes, and reports information with Runtime Status and Control Application (RSCA) Programming Interface.
`TracingModule`	Reports events to Microsoft Event Tracing for Windows (ETW).
Managed Support Modules	
`ManagedEngine`	Provides integration of managed code modules in the IIS request-processing pipeline.
`ConfigurationValidationModule`	Validates configuration issues, such as when an application is running in Integrated mode but has handlers or modules declared in the system.web section.

The preceding modules are all installed with a full installation of IIS 7.0. However, if IIS 7.0 is installed with the default configuration and modules, a subset of those modules are installed. The modules installed by default are listed as follows.

- HTTP modules
  - `CustomErrorModule`
  - `ProtoclSupportModule`
- Security modules
  - `RequestFilteringModule`
  - `AnonymousAuthenticationModule`
- Content modules
  - `DefaultDocumentModule`
  - `DirectoryListingModule`
  - `StaticFileModule`
- Content modules
  - `StaticCompressionModule`
- Logging and diagnostics modules
  - `HTTPLoggingModule`
  - `RequestMonitorModule`
- Caching modules
  - `HttpCacheModule`

Managed Modules

IIS 7.0 infrastructure allows the installation of .NET managed modules to participate in the request-processing pipeline. Allowing managed modules to function properly depends mostly on the `ManagedEngineModule` mentioned above. Managed modules are ASP.NET 2.0 and 3.5 `HttpModules` that a .NET developer has always been used to writing, however with IIS 7.0, these modules will get the chance to work upon requests during the request-processing pipeline managed by the web server itself.

The existing managed modules that can be configured with IIS 7.0 are listed in the following table.

Module Name	Description
`AnonymousIdentification`	Manages anonymous identifiers, which are used by features that support anonymous identification such as ASP.NET profile engine.
`DefaultAuthentication`	Ensures that an authentication object is present in the context.
`FileAuthorization`	Verifies that a user has permission to access the requested file.

Continued

Module Name	Description
FormsAuthentication	Supports authentication by using Forms authentication.
OutputCache	Supports output caching
Profile	Manages user profiles by using ASP.NET profile, which stores and retrieves user settings in a data source such as a database.
RoleManager	Manages a RolePrincipal instance for the current user.
Session	Supports maintaining session state, which enables storage of data specific to a single client within an application on the server.
UrlAuthorization	Determines whether the current user is permitted access to the requested URL, based on the user name or the list of roles that a user is member of.
UrlMappingsModule	Supports mapping a real URL to a more user-friendly URL.
WindowsAuthentication	Sets the identity of the user for an ASP.NET application when Windows authentication is enabled.

This managed modules' information has been gathered from the official ASP.NET 2.0/3.5 documentation on MSDN.

Summary

In this chapter you were introduced to the new web server engine by Microsoft, IIS 7.0. IIS 7.0 ships with a new architecture that is more modular and allows administrators and developers to configure it the way they want.

The main point to keep in mind about the new web server is its modular architecture. IIS 7.0 is installed with minimal modules or features. Additional modules can be installed whenever they are needed. In addition, IIS 7.0 allows developing both native and managed modules using C++ and .NET, respectively.

A lot of improvements have been introduced to IIS 7.0, including security, administration and configuration, and troubleshooting improvements. New APIs are now ready for use by native and managed developers to extend the functionality of the web server.

IIS 7.0 now integrates well with ASP.NET infrastructure for request processing; hence, applications now can run either in the Integrated mode or in the Classic mode application pool.

- **Integrated mode:** When running under the Integrated mode, the ASP.NET 2.0 or 3.5 application can take benefit from the integration between IIS 7.0 and ASP.NET so that a single unified pipeline is present where both IIS native modules and configured ASP.NET modules have a say while processing a specific request.
- **Classic mode:** With the Classic mode, an application will have the same environment as it had once under IIS 6.0, where the IIS 7.0 request-processing pipeline happens separately from the ASP.NET request-processing pipeline.

In addition, IIS 7.0 components have been enhanced and a new major component that has been added is the Windows Process Activation Service (WAS). This service is the brain of the web server that interacts with the web server configuration system and configures protocol listener adapters that in turn configure their corresponding protocol listeners. This new service handles both HTTP and non-HTTP requests, and this gives IIS a broader field to handle so many requests from different sources. Also, this service is responsible for the process management including application pool states, stopping, starting, recycling them, and creating new worker process instances.

The next chapter continues this discussion with a look at the new IIS 7.0 and ASP.NET Integrated mode. The discussion includes a thorough examination of the Integrated mode architecture as well as developing new modules and handlers in ASP.NET and integrating them with IIS 7.0 infrastructure. In addition, a study on handling migration errors is given to help in migrating an existing ASP.NET application to run under the IIS 7.0 and ASP.NET Integrated mode.